PushUp: A Scalable Event-driven Long-Polling Server

Kai Liu
Very Large Information System
Carnegie Mellon University
Pittsburgh, PA
hfevers@gmail.com

Yilun Cui
Very Large Information System
Carnegie Mellon University
Pittsburgh, PA
yiluncui@gmail.com

ABSTRACT

The "long-polling" is a server push technology that makes the delivery the latest updates from server to client more "instant". With long polling, when a client requests updates from the server, the server will keep the request connection open if there is no available update yet. The server sends the complete response to the server only when the interested information becomes available or after a suitable timeout.

However this solution may require the server to hold large number of open connections for in the server side. To efficiently manage these connections and reduce the complexity of web development, we proposed an event-driven, dedicated long-polling server to provide efficient and scalable long polling services for the underlying web servers. To prove its efficiency, we compare the performance of PushUp server and several other server-side event notification methods. The experimental results confirms the PushUp server's excellent capacity in serving large number of concurrent connections with much less resource.

1. INTRODUCTION

Traditionally, the web server is a classical example of the request/response model, where the clients can retrieve the resource from the server but the server cannot push the data to the client directly.

However, with the increasing popularity of Ajax[3] technologies, there are many scenarios where servers would like to actively notify the clients the latest updates. For example, the new messages in chat room, latest tweets in twitter, etc.

To overcome the limitation of request/response model, many approaches has been proposed to build more responsive sites. For example:

- Client Pull: the client sends update requests periodically to pull the updates from server. This method is very easy to implement because it is stateless and the web server doesn't need to put much effort to modify its existing architecture. However, client pull will generate lots of http requests. The work flow of client pull is illustrated in Figure 1.

- Plugins: since the HTTP itself doesn't provide any mechanism for server push, plugins, such as the Flash, Silverlight, Javelet, etc, can be used as an extension of the HTTP by providing more flexible client/server communication mechanisms. However plugins is non-standard and requires users' extra effort to install and configure them.

Given those drawbacks, we consider another solution: long polling [5]. The work flow of long polling can be described in Figure 2. When a client "asks for"(polls) an update by sending an HTTP request to the server, if no information is yet available, the server will not terminate the connection immediately; Instead, the server will keep the connection open for a period of time until (1) the information become available (2) after a suitable timeout(50 seconds is a common choice in many commercial websites).

The benefit to long-polling is that there is less back-and-forth between the client and server. The server is in control of the timing, so updates to the browser can normally be made within milliseconds. This makes it ideal for some highly interactive web applications.

The down-side of long polling is that the server may have to deal with large amount of active connections between the clients and the servers. For example, suppose a website has one million users and, 10% of them will be online. In this case, the server should be able to hold at least 100,000 concurrent connections.

To address these problems, we present the event-driven[12], scalable long polling server PushUp to provide dedicated long polling services for different web applications with less resource cost.

As illustrated in Figure 3, the PushUp server, designed as a dedicated long polling server, introduces an intermediary between the clients and web servers. Inside the PushUp server there is a specialized event-driven message queue that:

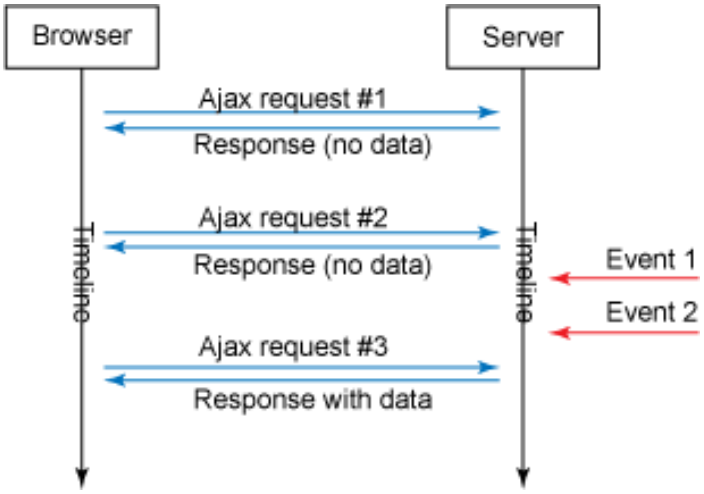

Figure 1: Client Pull Work Flow

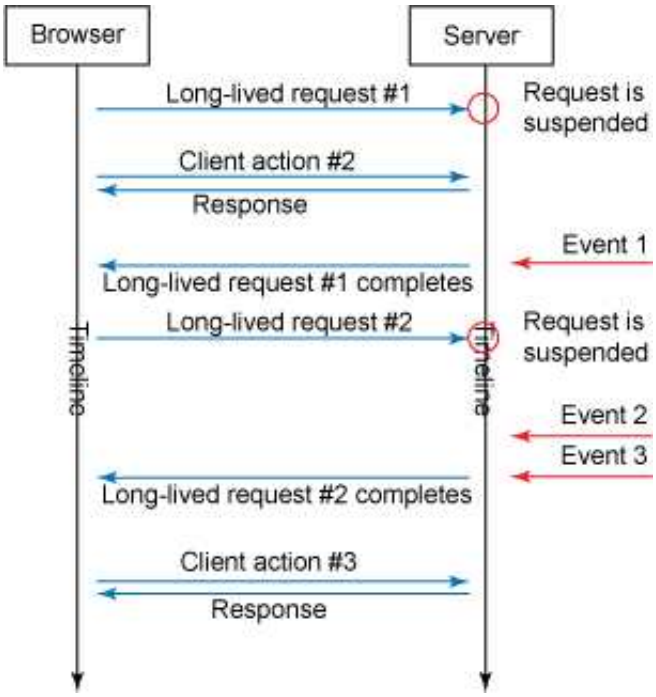

Figure 2: Long Polling Work Flow

1 Keeps the active connections with low overhead.

2 Provides a familiar interface for clients to listen to the latest updates and the web servers to notify the newest changes.

2. SYSTEM OVERVIEW

2.1 Design Goals and Rationales

The design of the PushUp server is driven by the following rationales.

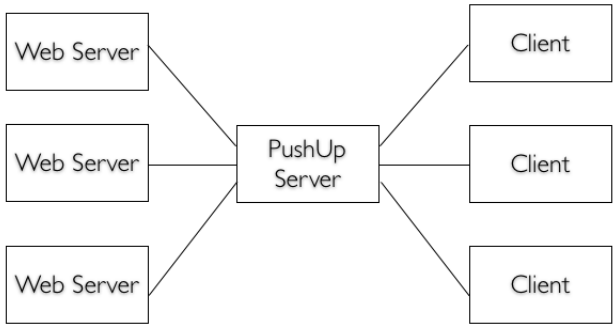

Figure 3: Simplified Architecture of the PushUp

- **Large Concurrent Connections**: As the long polling technologies requires the server to keep the polling request active for a period of time, large number of concurrent clients will generate lots of active connections. One of the most goal of PushUp is to minimize the cost of maintaining these active concurrent connections, which can in turn enhance the system's scalability.

  The key to reduce the cost is the event-driven concurrency architecture. Unlike the multiprocessing or multithreading models, the event-driven model requires much less resource to maintain the connections, avoids context switches between threads and expensive lock operations.

- **Generality**: Although event-driven concurrency significantly reduce the maintenance cost, it is impractical to have all the web server to be implemented in event-driven programming style. Thus, it makes sense to offer a dedicated long polling services, which enables all the web servers to obtain the long polling functionality with minimal effort.

  Moreover a general long polling service provider prevents web servers from re-inventing the wheels and, according the Linus Law[6] " Many eyes make bugs shallow", helps to make the service more robust.

- **Low Latency**: Adding an intermediate layer in between of the clients and web servers will inevitably create some overhead. The PushUp server should make sure that the extra costs be controlled within an "acceptable" range. In the evaluation section we will carefully examine the latency of the PushUp servers.

- **Scalability**: The PushUp Server is designed to handle large number of active connections. In later sections we well further explore mechanisms that help the PushUp scales better.

2.2 System Architecture

Figure 4 shows the overall architecture of the PushUp servers. The PushUp server is essentially a specialized message queue, with clients acting as the subscribers and the web server the publishers.

When a client initites an http request, the reverse proxy acts as the web front end, which will:

1 Forwards the normal(non-polling) http requests to the corresponding web servers.

2 Intercepts the polling http requests and sends a subscription request to the PushUp servers.

The reverse proxy is a kind of proxy that retrieves resources on behalf of a client from one or more servers.[7]. In this case, the reverse proxies hide the existence of the origin servers (the web servers and PushUp servers) and presents the clients a uniform interface.

On the other end, the web servers can send the publication requests to the PushUp servers in order to inform interestd clients with the latest updates.

As suggested figure 4, there can be more than one PushUp servers working together. Both subscription and publication will go through a load balancer, which distributes these requests to one of the servers. All PushUp servers are "identical" to each other. That is, a subscriber can subscribe to any of PushUp servers and the publisher can publish message to whatever PushUp server. We will discuss more details in next section.

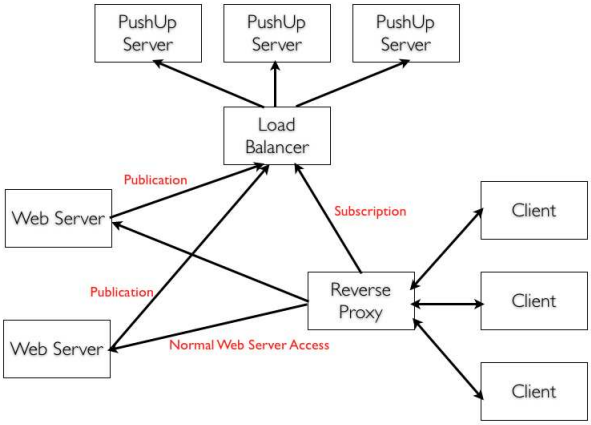

Figure 4: PushUp Architecture

3. DESIGN AND IMPLEMENTATION

3.1 Reactor Model

The PushUp server runs atop of the Twisted framework[9], an event driven framework that implements the "reactor" style event loop.

The reactor patterns is an event processing pattern for handling concurrent service requests. It demultiplexes these requests to associated event handlers(implemented as the callback). The event handler performs the actual read or write synchronously.

In our project the reactor manages event loops (one Reactor can start several event loops) and runs in a single thread. The Twisted's event loop has well encapsulated the OSes' underlying event notification mechanism (for example, kqueue() in FreeBSD and epoll() in Linux). During execution, the event loop manages the details regarding to event handler registrations, the activations of event handlers, etc.

The advantage of the reactor patter is the complete separation of the application specific code from the reactor implementation. That is to say, the application components can be easily divided into reusable modulars. Also, running the reactor in a single threads significantly reduce the concurrency controls since no other threads will contend the system resource instantaneously.

Figure 5 demonstrates the how the execution switches between the event loops and user code. Figure 6 further illustrates the detailed work flow with sequence diagram.

As we can see from these figures, one of the caveats of in event-driven development is that all the event handlers should be carefully designed in order to make sure its execution time will not take a long time; otherwise the whole system will be blocked.

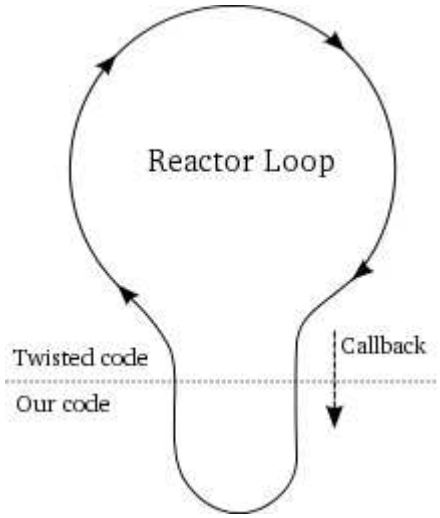

Figure 5: Work flow of Twisted Reactor Model

3.2 Channels

A "channel" represents message stream of a certain type. For example, a channel may be used to represent the feed of a specific user's tweets in twitter, or the articles of a blog. The channel is the building block of PushUp's message queue.

The channel has two major responsibilities: manages the subscription and incoming messages in this channel.

A complete subscription includes the following information:

- **Timeout(required)**: the channel will keep the subscription connection open only for a certain timespan.

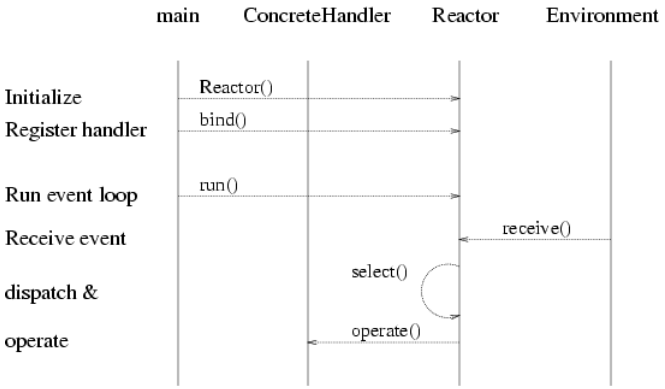

Figure 6: The Sequence Diagram of the Event Loop

If nothing happens during this period, the subscription request will be terminated and removed from the active subscription list by the channel.

- **Time Range(optional)**: the client may specify the "start time" and "end time" of its interested messages. Since the channel caches all the incoming messages for a certain time(we will explain this later), proper time range should be specified otherwise the client will always receive all the available messages in the channel (this is because if the time range is not specified, by default the channel will set the start time to be 0 and the end time to be *max_int*).

  Owing to the stateless nature of channel, the client should maintain and update its own start time and end time. Here is an example shows the basic use case of the time range: every time when a client receives the latest messages from server they will update the 'start-time' in the next polling request (or the subscription request in message queue's prospective). Thus, the clients can always fetch the "fresh" messages.

Now we'll discuss more about the subscription management and message management.

- **Subscription**: when a subscription arrives, the channel will check if there are any interested messages available. If there are, the channel will deliver the interested messages to the client immediately(and then close the connection); otherwise the channel will keep the subscription connection open in order to wait for latest messages. The channel may also force to close a subscription connection if it times out.

- **Messages**: After a message is published and "current" subscriptions are notified, the message will continue to reside in the message queue for a certain time (this time is configurable).

  The motivation of this design is that, unlike traditional message queue where the client can set up "one" persistent subscription connection to fetch the latest updates continuously, current Ajax technologies cannot start processing the server response until the connection is closed. As a result, the client has to initiate the polling requests again after it receives new messages (or timeout).

  A problem is thus raised: what if a message is published when a client is in the gap of "just received new messages" and "initiate a new subscription request"? The message caching ensures the client will not miss the messages published during that "gap".

The subscriptions and messages are organized by the B-trees[2], where timestamps are the keys and the subscriptions(or the messages) are the values. We choose B-tree because it provides good time complexity in range operations ($O(logN)$), which will facilitate the removal of expired subscriptions information(or messages), retrieval of the messages of a specific time range, etc.

3.3 Message Queue

The message queue manages the channels and presents the publish/subscribe interfaces for the external modules.

The message queue has the following responsibilities:

- **Create, Renew and Close a Channel**: the publisher can create a channel for its own use. After the *create* operation, the publisher will receive a unique channel id and the expired time.

  When a channel is inactive (no new messages published) for a while, the message queue will close this channel. Of course, The publisher can call *renew* operation to extend the expire time for that channel. The publisher can also call *close* operation to explicitly close that channel.

- **Batch Publication**: The publisher can publish messages to more than channels. The newly published message will be assigned with a unique id.

  For example, in a Q/A community, when a user John posts a new questions with the tags 'c++', 'java'. The questions will be published to the channel "author: John"[1], "tag:c++" and "tag:java".

- **Batch Subscription**: Like publisher, the subscriber can also specify more than one channels to listen. When messages are retrieved from several channels, the message queue will make sure all these messages are sorted by time and removes duplicated messages (because different channels may have the same message)

3.4 Communication between PushUp Servers

As we have already discussed in the system architecture, several PushUp servers may work together to achieve a better performance, as shown in figure 4.

The published messages will be available to all PushUp servers while the subscriptions will be distributed to several machines. For subscribers and publisher, it make no difference

[1]Actually the channel id is an integer. The textual channel id is used only to help readers better understand the work flow.

which PushUp server they interact with. The PushUp server works together to present a uniform interface. A subscriber can subscribe to any of PushUp servers and the publisher can publish message to whatever PushUp server.

The PushUp servers coordinate with each other in the following ways:

- **Publication**: When a PushUp server receives a publication request, it will broadcast that publication request to the other PushUp servers. Doing this allows all the interested subscriptions(across different machines) to be notified.
- **Subscription**: The subscription will simply be distributed to different PushUp servers. It is the load balancer's responsibility to make sure subscriptions will be dispatched evenly.

The motivation of this design is based on the observation that in a typical web server, the number of the subscriptions will be significantly larger than that of the publications.

4. EVALUATION

In this section, we conducted experiments to evaluate the performance of several web-based event[2] notification mechanisms. We also evaluated PushUp servers' performance with different server numbers.

To better characterize the performance of the PushUp server, we chose "Client Pulling" and "Multithreadeding long polling" as the baseline. The PushUp adopted the Event-driven long polling mechanism.

- **Client Pulling (CP):** In this method the client side initiates an http request to the server to pull the latest updates on a regular basis.
- **Multithreadeding long polling(MLP):** This method realizes the long polling by multi threads. For each incoming polling request, the server will create a thread waiting for the latest updates.
- **Event-driven long polling(ELP):** This method realizes the long polling by event-driven mechanism. This method keeps all the polling requests within a single thread.

4.1 Settings

4.1.1 Evaluation Metrics

In the evaluation, we used the following control variables to control in the experiments.

[2]the term "event" here refers to the "latest updates in the server side", which differs from the meaning in "event-driven"

- **Number of concurrent connections**: This variable allows us to evaluate the scalability of different methods.
- **Publish Interval**: This variable controls the frequency of event publication by the server.
- **Polling interval**: This variable determines how long will the method send a polling request to the server. For client pulling, it sends polling request at a regular basis; as for long polling methods, they send polling request (1) after they just received a new update or (2) after a suitable timeout.

And we used the following metrics to evaluate the performance of different methods.

- **Network Traffic (NT)**: NT evaluates the network usage for different event notification methods.
- **Mean Publish Trip Time (MPT)**: The publish trip-time is the time elapsed between the creation of a data by the server and its reception by the client. It shows how long it takes for the client to be updated when an event occurs.
- **Received Message Percentage (RMP)**: RMP indicates the data loss in during the network communication.
- **Server Memory Usage (SMU)**: In many real world long polling system, the polling requests spent most of their time waiting for the new events.[source needed] Thus, memory cost for each idle polling requests is critical to the overall performance.

4.1.2 Evaluation Environment

The experiments are done in CMU's PEACE cluster. Each machine in the cluster has a 4 CPU cores(AMD Opteron Processor 4130, 2600 MHz, 512K cache size), 8 GB RAM and 1 physical 581 GB disk.

4.2 Push Model vs. Poll Model

One of the biggest advantage of long polling strategy is its significant reduction of network traffic. In the following experiments we focused on the evaluation on the network traffic(NT) and mean publish trip time(MPT).

Since the goal of these experiments is to characterize the network traffic of different methods. We evaluated the **network traffic** and **mean publish trip time** only with some moderate number of concurrent connections: 10, 100 and 1000.

The **publish time** are $10sec$, $20sec$, $30sec$, $40sec$ and $50sec$. The **polling time** we chose for the **client pulling** is $1sec$, and $50sec$ for long polling based methods(which is a popular choice in real world long polling applications).

Table 1 shows the detailed network traffic with different publish time and number of concurrent connections (written as

Table 1: Network Traffic(KB)

cc=10	10 sec	20 sec	30 sec	40 sec	50 sec
CP	12.7	25.8	39.8	55.2	68.2
MLP	1.27	1.27	1.27	1.27	1.27
ELP	1.27	1.27	1.27	1.27	1.27
cc=100	10 sec	20 sec	30 sec	40 sec	50 sec
CP	127.4	277.1	415.2	580.5	745.5
MLP	12.7	12.7	12.7	12.7	13.1
ELP	12.7	12.7	12.7	12.7	12.7
cc=1000	10 sec	20 sec	30 sec	40 sec	50 sec
CP	1370.5	3457.1	5537.5	7282.5	10743.5
MLP	135.5	132.5	137.8	133.4	135.5
ELP	130.1	129.5	132.8	131.8	130.5

Table 2: Mean Publish Trip Time (second)

cc=10	10 sec	20 sec	30 sec	40 sec	50 sec
CP	0.6	0.7	0.5	0.7	0.8
MLP	0.1	0.0	0.1	0.2	0.1
ELP	0.2	0.1	0.1	0.4	0.2
cc=100	10 sec	20 sec	30 sec	40 sec	50 sec
CP	0.8	1.1	2.5	3.1	2.7
MLP	0.5	0.4	0.5	0.8	1.1
ELP	0.7	0.4	0.8	1.2	0.9
cc=1000	10 sec	20 sec	30 sec	40 sec	50 sec
CP	2.8	3.4	4.5	5.3	7.2
MLP	2.3	2.9	2.7	3.5	3.4
ELP	2.4	2.8	1.8	2.4	2.9

"cc" in the tables). We can see that the long polling based method keeps a relatively constant network traffic while the client pull method's network traffic grows (almost) linearly as the publish time increases. Figure 7 shows the network traffic change when the number of concurrent connection is 1000.

Table 2 demonstrates the mean publish trip time with different publish time and number of concurrent connections. The MPT of client pull increases much faster than the long polling based methods. This may because the growing network traffic increases the chance of package loss in the client pull method. Figure 8 compares the MPT of push and pull models when the number of concurrent connection is 1000.

In summary, the push model significantly reduce the network traffic, which in turn, yields a smaller mean publish trip time as the concurrent connections increase.

Also, we observed that the multithreading-based long polling and event-driven long polling have similar performance in terms of network traffic and mean publish trip time when the number of concurrent connections is relatively small. In the next evaluation we will test both long polling methods' performance with large concurrent connections.

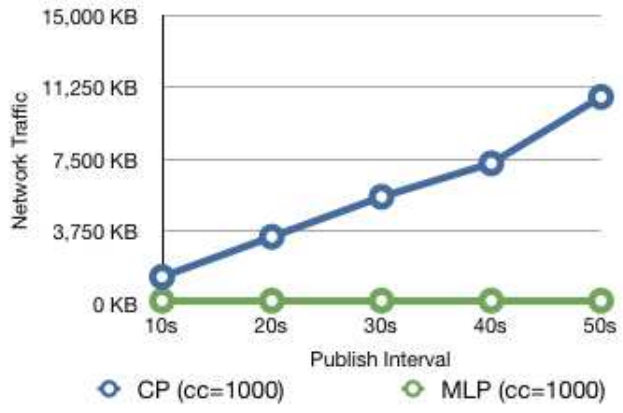

Figure 7: Network Traffic of Pull & Push Models

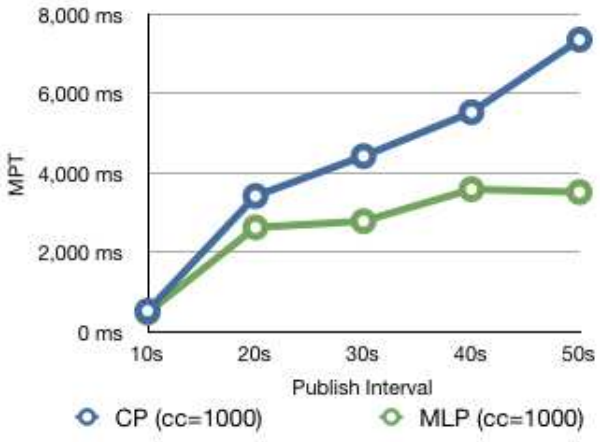

Figure 8: Mean Publish Trip Time of Pull & Push Models

4.3 Event vs. Thread

Here we compared the performance between event-driven long polling method and the multithreading long polling method. In the following evaluation we tested the **server memory usage(SMU)**, **mean publish trip time (MPT)**, **received message percentage(RMP)** with large number of active connections(from 500-16000).

Figure 9 shows the server memory usage of both methods. The event-driven method demonstrates an excellent memory usage as the number of active connections increases, while the memory usage of multithreading method soars rapidly and reaches 100% when the active connections' number goes to 8000.

Figure 10 and figure 11, show the mean publish trip time and received message percentage. Event-driven method significantly excels multithreading method in both evaluation.

Thus we claimed one-polling-request-per-thread approach is not a good choice for handling large amount of concurrent connections, as its memory usage and possible context switch may incur very large overhead. The event-driven

method, on the other hand, demonstrates a better performance in memory usage and hosting large concurrency.

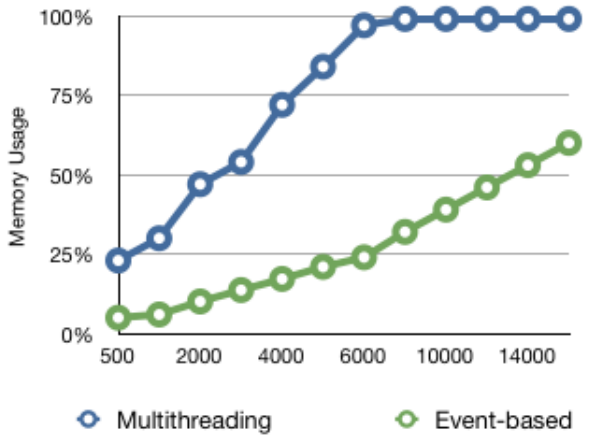

Figure 9: Server Memory Usage

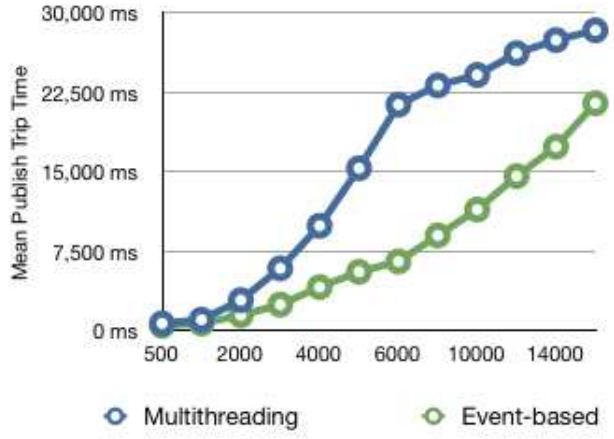

Figure 10: Mean Publish Trip Time

4.4 Extending PushUp Server

As we saw the advantages of PushUp over other event notificaition methods, now we would like to extend one PushUp server to 2, 3 and 4 servers and see the performance changes as PushUp scales.

In this evaluation, the incoming polling requests will be (almost) evenly distributed to each PushUp server. A simple round-robin will be sufficient for this task. We used the HAProxy[8] as a load balancer, which simply balances the load in the TCP layer.

Figure 12 and figure 13, respectively, show the **mean publish trip time** and **received message rate** of the PushUp servers with different nodes.

These figures verify that the PushUp servers scale almost linearly as the number of nodes increases, with a very small overhead in load balancing and updates propagation among PushUp servers.

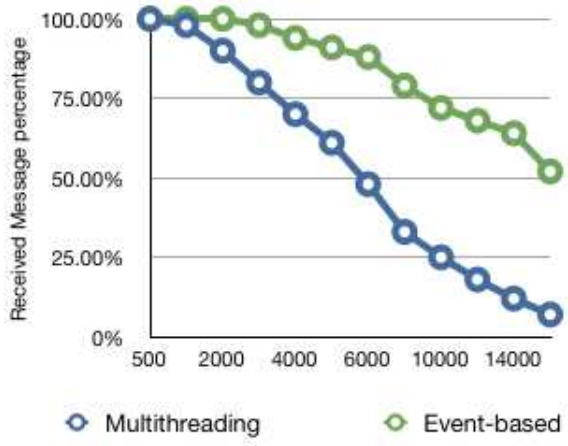

Figure 11: Received Message Percentage

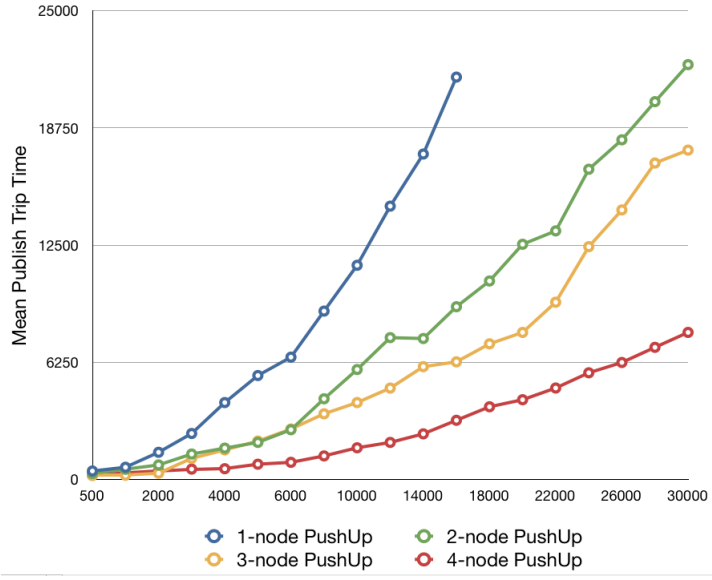

Figure 12: Mean Publish Trip Time

5. RELATED WORK

5.1 Event-driven Systems

The paper[11] evaluated several strategies to achieve concurrency for web servers. The authors compared performance of different concurrent models: multi-process, multi-thread, event-driven, etc. As the experimental results reveal, the event-driven μ-server performed better than multithreading or multiprocessing models

In the famous technical report[10], Kegel performed detailed benchmark on several I/O strategies. The report compares

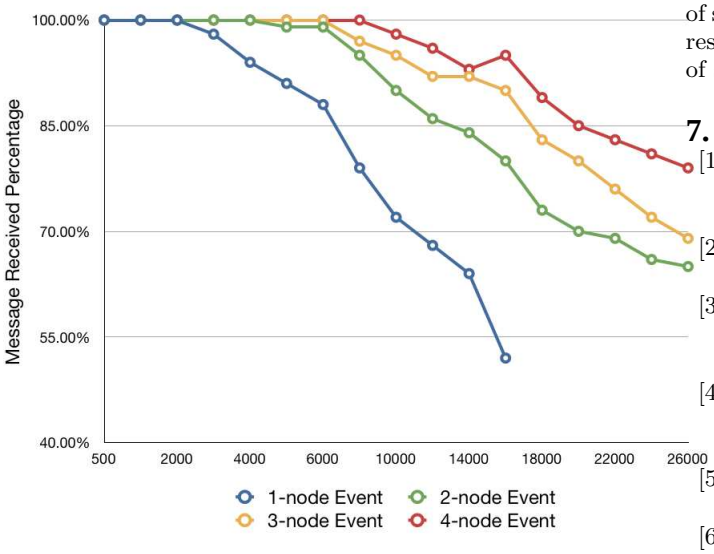

Figure 13: Received Message Percentage

different system supports for the I/O multiplexing(which are the foundation of event-driven programming): select(), poll(), epoll() and kqueue() and argues that the implementation of epoll() and select() makes them scales poorly while the epoll() and kqueue() performs much better as they avoid active pollings to the system kernels.

5.2 Server Push technologies

In [1], Bozdag *et al* provided complete comparisons and evaluation of different types of server push technologies, including the HTTP pull, Flash XML socket, Java RMI, long polling etc. As a conclusion the authors stated that "an event-driven model on the server side is necessary" owing to its , lower consumption of the computing resource, less network traffic etc.

Another paper [4] performs the quantitative analysis of the consistency and scalability of several different server push technologies. As the experimental results showed the event-driven long polling model shows an excellent efficiency and scalability.

6. SUMMARY AND CONCUSION

In this paper we examined an event-driven scalable long polling server PushUp. As a dedicated long polling server, the aim of the PushUp is to be efficiently handle large number of concurrent connections and provides general long polling services for different backend web servers.

We compared the PushUp server's performance with other event notification models. The experimental results demonstrate that excellent capability of PushUp server in terms of significantly reduced network traffic, lower memory usage, slowly growing latency (as the number of concurrent connections gets larger), etc. To further evaluate the scalability the PushUp Server, we also tested the performance of several collaborating PushUp servers, whose experimental results shows that PushUp servers scale well as the numbers of server increases.

7. REFERENCES

[1] E. Bozdag, A. Mesbah, and A. V. Deursen. A Comparison of Push and Pull Techniques for Ajax. *Computing Research Repository*, abs/0706.3, 2007.

[2] D. E. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11:121–137, 1979.

[3] C. Draganova and V. Vassilev. Teaching AJAX in web-centric courses. *ACM Sigcse Bulletin*, 39:311–311, 2007.

[4] S. Duquennoy, G. Grimaud, and J.-J. Vandewalle. Consistency and scalability in event notification for embedded Web applications. Sept. 2009.

[5] http://en.wikipedia.org/wiki/Comet_(programming). Comet Programming.

[6] http://en.wikipedia.org/wiki/Linus'_Law. Linus's Law.

[7] http://en.wikipedia.org/wiki/Reverse_proxy. Reverse Proxy.

[8] http://haproxy.1wt.eu/. HAProxy Official Website.

[9] http://www.twistedmatrix.com/. Twisted Matrix Lab.

[10] D. Kegel. The C10K problem. Sept. 2006.

[11] D. Pariag, T. Brecht, A. Harji, P. Buhr, A. Shukla, and D. R. Cheriton. Comparing the performance of web server architectures. *Operating Systems Review*, 41:231–243, 2007.

[12] W. R. Stevens and T. Narten. Unix network programming. 20:8–9, 1990.